AD-A242 284

REPORT DOCUMENTATION PAGE

Form Approved
OPM No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
		Final: 02 Nov 1990 tp 01 Jun 1993

4. TITLE AND SUBTITLE
901102W11055Alsys, Inc., AlsyCOMP_016 Version 5.1, CompuAdd 320 (Host & Target), 901102W1.11056

5. FUNDING NUMBERS

6. AUTHOR(S)
Wright-Patterson AFB, Dayton, OH
USA

DTIC

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Ada Validation Facility, Language Control Facility ASD/SCEL
Bldg. 676, Rm 135
Wright-Patterson AFB, Dayton, OH 45433

8. PERFORMING ORGANIZATION REPORT NUMBER
AVF-VSR-410-0891

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
United States Department of Defense
Pentagon, Rm 3E114
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
No software available for distribution per Michelle Kee, ADA, telecon 11/4/91

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
Alsys, Inc., AlsyCOMP_016 Version 5.1, Wright-Patterson AFB, OH, CompuAdd 320 (Host & Target), ACVC 1.11.

91-15050

14. SUBJECT TERMS
Ada programming language, Ada Compiler Val. Summary Report, Ada Compiler Val. Capability, Val. Testing, Ada Val. Office, Ada Val. Facility, ANSI/MIL-STD-1815A, AJPO.

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFED	UNCLASSIFIED	

NSN 7540-01-280-550

Standard Form 298, (Rev. 2-89)
Prescribed by ANSI Std. 239-128

91 1104 120

AVF Control Number: AVF-VSR-410-0891
23-August-1991
90-06-01-ALS

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 901102W1.11056
Alsys, Inc.
AlsyCOMP_016 Version 5.1
CompuAdd 320 => CompuAdd 320

Prepared By:
Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

A-1

Certificate Information

The following Ada implementation was tested and determined to pass ACVC 1.11. Testing was completed on 2 November 1990.

Compiler Name and Version: AlsyCOMP_016, Version 5.1

Host Computer System: CompuAdd 320 (under MS/DOS 3.30, Phar Lap 2.0)

Target Computer System: CompuAdd 320 (under MS/DOS 3.30, Phar Lap 2.0)

Customer Agreement Number: 90-06-01-ALS

See Section 3.1 for any additional information about the testing environment.

As a result of this validation effort, Validation Certificate 901102W1.11056 is awarded to Alsys, Inc. This certificate expires on 1 June 1993.

This report has been reviewed and is approved.

Steven P. Wilson

Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Ada Validation Organization
Director, Computer & Software Engineering Division
Institute for Defense Analyses
Alexandria VA 22311

Chris Denny

for

Ada Joint Program Office
Dr. John Solomond, Director
Department of Defense
Washington DC 20301

DECLARATION OF CONFORMANCE

CUSTOMER: Alsys, Inc.

ADA VALIDATION FACILITY: Ada Validation Facility (ASD/SCEL)
Computer Operations Division
Information Systems and Technology Center
Wright-Patterson AFB 0H 45433-6503

ACVC VERSION: 1.11

ADA IMPLEMENTATION:

COMPILER NAME AND VERSION: ALSYS_COMP_C16-386

HOST COMPUTER SYSTEM: CompuAdd 320
under MS/DOS 3.30, Phar Lap 2.0

TARGET COMPUTER SYSTEM: CompuAdd 320
under MS/DOS 3.30, Phar Lap 2.0

CUSTOMER'S DECLARATION:

I, the undersigned, representing Alsys, Inc., declare that Alsys, Inc. has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation listed in this declaration.

Mike Blanchette

90-oct-24

Mike Blanchette,
Vice President, Engineering
Alsys, Inc.
67 South Bedford Street
Burlington, MA 01803-5152

Date

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

The Ada implementation described above was tested according to the Ada Validation Procedures [Pro90] against the Ada Standard [Ada83] using the current Ada Compiler Validation Capability (ACVC). This Validation Summary Report (VSR) gives an account of the testing of this Ada implementation. For any technical terms used in this report, the reader is referred to [Pro90]. A detailed description of the ACVC may be found in the current ACVC User's Guide [UG89].

1.1 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the Ada Certification Body may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject implementation has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from the AVF which performed this validation or from:

National Technical Information Service
5285 Port Royal Road
Springfield VA 22161

Questions regarding this report or the validation test results should be directed to the AVF which performed this validation or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.2 REFERENCES

[Ada83] Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

[Pro90] Ada Compiler Validation Procedures, Version 2.1, Ada Joint Program Office, August 1990.

[UG89] Ada Compiler Validation Capability User's Guide, 21 June 1989.

1.3 ACVC TEST CLASSES

Compliance of Ada implementations is tested by means of the ACVC. The ACVC contains a collection of test programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable. Class B and class L tests are expected to produce errors at compile time and link time, respectively.

The executable tests are written in a self-checking manner and produce a PASSED, FAILED, or NOT APPLICABLE message indicating the result when they are executed. Three Ada library units, the packages REPORT and SPPRT13, and the procedure CHECK_FILE are used for this purpose. The package REPORT also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The package SPPRT13 is used by many tests for Chapter 13 of the Ada Standard. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. If these units are not operating correctly, validation testing is discontinued.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that all violations of the Ada Standard are detected. Some of the class B tests contain legal Ada code which must not be flagged illegal by the compiler. This behavior is also verified.

Class L tests check that an Ada implementation correctly detects violation of the Ada Standard involving multiple, separately compiled units. Errors are expected at link time, and execution is attempted.

In some tests of the ACVC, certain macro strings have to be replaced by implementation-specific values -- for example, the largest integer. A list of the values used for this implementation is provided in Appendix A. In addition to these anticipated test modifications, additional changes may be required to remove unforeseen conflicts between the tests and implementation-dependent characteristics. The modifications required for this implementation are described in section 2.3.

For each Ada implementation, a customized test suite is produced by the AVF. This customization consists of making the modifications described in the preceding paragraph, removing withdrawn tests (see section 2.1) and, possibly some inapplicable tests (see Section 2.2 and [UG89]).

In order to pass an ACVC an Ada implementation must process each test of the customized test suite according to the Ada Standard.

1.4 DEFINITION OF TERMS

Ada Compiler	The software and any needed hardware that have to be added to a given host and target computer system to allow transformation of Ada programs into executable form and execution thereof.
Ada Compiler Validation Capability (ACVC)	The means for testing compliance of Ada implementations, consisting of the test suite, the support programs, the ACVC user's guide and the template for the validation summary report.
Ada Implementation	An Ada compiler with its host computer system and its target computer system.
Ada Joint Program Office (AJPO)	The part of the certification body which provides policy and guidance for the Ada certification system.
Ada Validation Facility (AVF)	The part of the certification body which carries out the procedures required to establish the compliance of an Ada implementation.
Ada Validation Organization (AVO)	The part of the certification body that provides technical guidance for operations of the Ada certification system.
Compliance of an Ada Implementation	The ability of the implementation to pass an ACVC version.
Computer System	A functional unit, consisting of one or more computers and associated software, that uses common storage for all or part of a program and also for all or part of the data necessary for the execution of the program; executes user-written or user-designated programs; performs user-designated data manipulation, including arithmetic operations and logic operations; and that can execute programs that modify themselves during execution. A computer system may be a stand-alone unit or may consist of several inter-connected units.

Conformity	Fulfillment by a product, process or service of all requirements specified.
Customer	An individual or corporate entity who enters into an agreement with an AVF which specifies the terms and conditions for AVF services (of any kind) to be performed.
Declaration of Conformance	A formal statement from a customer assuring that conformity is realized or attainable on the Ada implementation for which validation status is realized.
Host Computer System	A computer system where Ada source programs are transformed into executable form.
Inapplicable test	A test that contains one or more test objectives found to be irrelevant for the given Ada implementation.
ISO	International Organization for Standardization.
LRM	The Ada standard, or Language Reference Manual, published as ANSI/MIL-STD-1815A-1983 and ISO 8652-1987. Citations from the LRM take the form "<section>.<subsection>:<paragraph>."
Operating System	Software that controls the execution of programs and that provides services such as resource allocation, scheduling, input/output control, and data management. Usually, operating systems are predominantly software, but partial or complete hardware implementations are possible.
Target Computer System	A computer system where the executable form of Ada programs are executed.
Validated Ada Compiler	The compiler of a validated Ada implementation.
Validated Ada Implementation	An Ada implementation that has been validated successfully either by AVF testing or by registration [Pro90].
Validation	The process of checking the conformity of an Ada compiler to the Ada programming language and of issuing a certificate for this implementation.
Withdrawn test	A test found to be incorrect and not used in conformity testing. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains erroneous or illegal use of the Ada programming language.

CHAPTER 2

IMPLEMENTATION DEPENDENCIES

2.1 WITHDRAWN TESTS

The following tests have been withdrawn by the AVO. The rationale for withdrawing each test is available from either the AVO or the AVF. The publication date for this list of withdrawn tests is 12 October 1990.

E28005C	B28006C	C34006D	B41308B	C43004A	C45114A
C45346A	C45612B	C45651A	C46022A	B49008A	A74006A
C74308A	B83022B	B83022H	B83025B	B83025D	B83026B
B85001L	C83026A	C83041A	C97116A	C98003B	BA2011A
CB7001A	CB7001B	CB7004A	CC1223A	BC1226A	CC1226B
BC3009B	BD1B02B	BD1B06A	AD1B08A	BD2A02A	CD2A21E
CD2A23E	CD2A32A	CD2A41A	CD2A41E	CD2A87A	CD2B15C
BD3006A	BD4008A	CD4022A	CD4022D	CD4024B	CD4024C
CD4024D	CD4031A	CD4051D	CD5111A	CD7004C	ED7005D
CD7005E	AD7006A	CD7006E	AD7201A	AD7201E	CD7204B
BD8002A	BD8004C	CD9005A	CD9005B	CDA201E	CE2107I
CE2117A	CE2117B	CE2119B	CE2205B	CE2405A	CE3111C
CE3118A	CE3411B	CE3412B	CE3607B	CE3607C	CE3607D
CE3812A	CE3814A	CE3902B			

In addition to the tests indicated above, C35702A was withdrawn as a result of a challenge by this customer; it was included in the subsequent list of withdrawn tests, dated 21 November 1990.

2.2 INAPPLICABLE TESTS

A test is inapplicable if it contains test objectives which are irrelevant for a given Ada implementation. Reasons for a test's inapplicability may be supported by documents issued by ISO and the AJPO known as Ada Commentaries and commonly referenced in the format AI-ddddd. For this implementation, the following tests were determined to be inapplicable for the reasons indicated; references to Ada Commentaries are included as appropriate.

The following 201 tests have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y (14 tests)	C35705L..Y (14 tests)
C35706L..Y (14 tests)	C35707L..Y (14 tests)
C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)
C45421L..Y (14 tests)	C45521L..Z (15 tests)
C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)

The following 21 tests check for the predefined type LONG_INTEGER:

C35404C	C45231C	C45304C	C45411C	C45412C
C45502C	C45503C	C45504C	C45504F	C45611C
C45612C	C45613C	C45614C	C45631C	C45632C
B52004D	C55B07A	B55B09C	B86001W	C86006C
CD7101F				

C35713D and B86001Z check for a predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

C45531M..P and C45532M..P (8 tests) check fixed-point operations for types that require a SYSTEM.MAX_MANTISSA of 47 or greater; for this implementation, MAX_MANTISSA is less than 47.

C45536A, C46013B, C46031B, C46033B, and C46034B contain 'SMALL representation clauses which are not powers of two.

CD2A53A checks operations of a fixed-point type for which a length clause specifies a power-of-ten TYPE'SMALL; this implementation does not support decimal 'SMALLs. (See section 2.3.)

C45624A checks that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types with digits 5. For this implementation, MACHINE_OVERFLOWS is TRUE.

C45624B checks that the proper exception is raised if MACHINE_OVERFLOWS is FALSE for floating point types with digits 6. For this implementation, MACHINE_OVERFLOWS is TRUE.

C86001F recompiles package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete.

B86001Y checks for a predefined fixed-point type other than DURATION.

C96005B checks for values of type DURATION'BASE that are outside the range of DURATION. There are no such values for this implementation.

CD1009C uses a representation clause specifying a non-default size for a floating-point type.

CD2A84A, CD2A84E, CD2A84I..J (2 tests), and CD2A84O use representation clauses specifying non-default sizes for access types.

BD8001A, BD8003A, BD8004A..B (2 tests), and AD8011A use machine code insertions.

The tests listed in the following table are not applicable because the given file operations are supported for the given combination of mode and file access method.

Test	File Operation	Mode	File Access Method
CE2102E	CREATE	OUT_FILE	SEQUENTIAL_IO
CE2102F	CREATE	INOUT_FILE	DIRECT_IO
CE2102J	CREATE	OUT_FILE	DIRECT_IO
CE2102N	OPEN	IN_FILE	SEQUENTIAL_IO
CE2102O	RESET	IN_FILE	SEQUENTIAL_IO
CE2102P	OPEN	OUT_FILE	SEQUENTIAL_IO
CE2102Q	RESET	OUT_FILE	SEQUENTIAL_IO
CE2102R	OPEN	INOUT_FILE	DIRECT_IO
CE2102S	RESET	INOUT_FILE	DIRECT_IO
CE2102T	OPEN	IN_FILE	DIRECT_IO
CE2102U	RESET	IN_FILE	DIRECT_IO
CE2102V	OPEN	OUT_FILE	DIRECT_IO
CE2102W	RESET	OUT_FILE	DIRECT_IO
CE3102F	RESET	Any Mode	TEXT_IO
CE3102G	DELETE	--------	TEXT_IO
CE3102I	CREATE	OUT_FILE	TEXT_IO
CE3102J	OPEN	IN_FILE	TEXT_IO
CE3102K	OPEN	OUT_FILE	TEXT_IO

The tests listed in the following table are not applicable because the given file operations are not supported for the given combination of mode and file access method.

Test	File Operation	Mode	File Access Method
CE2105A	CREATE	IN_FILE	SEQUENTIAL_IO
CE2105B	CREATE	IN_FILE	DIRECT_IO
CE3109A	CREATE	IN_FILE	TEXT_IO

The following 16 tests check operations on sequential, direct, and text files when multiple internal files are associated with the same external file and one or more are open for writing; USE_ERROR is raised when this association is attempted.

CE2107B..E	CE2107G..H	CE2107L	CD2110B	CE2110D
CE2111D	CE2111H	CE3111B	CE3111D..E	CE3114B
CE3115A				

CE2111C raises a USE_ERROR when file is RESET from IN_FILE to OUT_FILE.

CE2203A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for SEQUENTIAL_IO. This implementation does not restrict file capacity.

EE2401D and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types with discriminants without defaults. These instantiations are rejected by this compiler.

CE2401H raises USE_ERROR when CREATE with mode INOUT_FILE is used for unconstrained records with default discriminants.

CE2403A checks that WRITE raises USE_ERROR if the capacity of the external file is exceeded for DIRECT_IO. This implementation does not restrict file capacity.

CE3304A checks that USE_ERROR is raised if a call to SET_LINE_LENGTH or SET_PAGE_LENGTH specifies a value that is inappropriate for the external file. This implementation does not have inappropriate values for either line length or page length.

CE3413B checks that PAGE raises LAYOUT_ERROR when the value of the page number exceeds COUNT'LAST. For this implementation, the value of COUNT'LAST is greater than 150000 making the checking of this objective impractical.

2.3 TEST MODIFICATIONS

Modifications (see section 1.3) were required for 25 tests.

The following tests were split into two or more tests because this implementation did not report the violations of the Ada Standard in the way expected by the original tests.

B23004A	B24007A	B24009A	B25002A	B26005A	B27005A
B28003A	B32202A	B32202B	B32202C	B37004A	B45102A
B61012A	B85002A	B91004A	B95069A	B95069B	B97103E
BA1101B	BC2001D	BC3009A	BC3009C		

BA2001E was graded passed by Evaluation Modification as directed by the AVO. The test expects that duplicate names of subunits with a common ancestor will be detected as compilation errors; this implementation detects the errors at link time, and the AVO ruled that this behavior is acceptable.

CD2A53A was graded inapplicable by Evaluation Modification as directed by the AVO. The test contains a specification of a power-of-10 value as small for a fixed-point type. The AVO ruled that, under ACVC 1.11, support of decimal smalls may be omitted.

EA3004D was graded passed by Evaluation and Processing Modification as directed by the AVO. The test requires that either pragma INLINE is obeyed for a function call in each of three contexts and that thus three library units are made obsolete by the re-compilation of the inlined function's body, or else the pragma is ignored completely. This implementation obeys the pragma except when the call is within the package specification. When the test's files are processed in the given order, only two units are made obsolete; thus, the expected error at line 27 of file EA3004D6M is not valid and is not flagged. To confirm that indeed the pragma is not obeyed in this one case, the test was also processed with the files re-ordered so that the re-compilation follows only the package declaration (and thus the other library units will not be made obsolete, as they are compiled later); a "NOT APPLICABLE" result was produced, as expected. The revised order of files was 0-1-4-5-2-3-6.

CHAPTER 3

PROCESSING INFORMATION

3.1 TESTING ENVIRONMENT

The Ada implementation tested in this validation effort is described adequately by the information given in the initial pages of this report.

For a point of contact for technical information about this Ada implementation system, see:

Mike Blanchette
67 South Bedford Street
Burlington MA 01803-5152

For a point of contact for sales information about this Ada implementation system, see:

Mike Blanchette
67 South Bedford Street
Burlington MA 01803-5152

Testing of this Ada implementation was conducted at the customer's site by a validation team from the AVF.

3.2 SUMMARY OF TEST RESULTS

An Ada Implementation passes a given ACVC version if it processes each test of the customized test suite in accordance with the Ada Programming Language Standard, whether the test is applicable or inapplicable; otherwise, the Ada Implementation fails the ACVC [Pro90].

For all processed tests (inapplicable and applicable), a result was obtained that conforms to the Ada Programming Language Standard.

a)	Total Number of Applicable Tests	3789
b)	Total Number of Withdrawn Tests	82
c)	Processed Inapplicable Tests	98
d)	Non-Processed I/O Tests	0
e)	Non-Processed Floating-Point Precision Tests	201
f)	Total Number of Inapplicable Tests	299
g)	Total Number of Tests for ACVC 1.11	4170

All I/O tests of the test suite were processed because this implementation supports a file system. The above number of floating-point tests were not processed because they used floating-point precision exceeding that supported by the implementation. When this compiler was tested, the tests listed in section 2.1 had been withdrawn because of test errors.

3.3 TEST EXECUTION

A magnetic tape containing the customized test suite (see section 1.3) was taken on-site by the validation team for processing. The contents of the magnetic tape were loaded onto a VAX/VMS system and then transferred to the host computer via a FTP LAN network.

After the test files were loaded onto the host computer, the full set of tests was processed by the Ada implementation.

Testing was performed using command scripts provided by the customer and reviewed by the validation team. See Appendix B for a complete listing of the processing options for this implementation. The options invoked explicitly for validation testing during this test were:

OPTION/SWITCH	EFFECT
SHOW => NO	Do not show header nor error summary in listing.
WARNING => NO	Do not include warning messages.
GENERIC => STUB	Place code of generic instantiation in separate subunits.
ERROR => 999	Maximum number of compilation errors permitted before terminating the compilation.
CALLS => INLINED	This option allows insertion of code for subprograms inline and must be set for the pragma INLINE to be operative.

TASK => 5	Size of all task stacks to be 5K bytes.
DIR => "-SYMB"	Directive for linker to generate low level debugger symbols.
MAIN => 100	Size of main stack to be 100K bytes.

Test output, compiler and linker listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

APPENDIX A

MACRO PARAMETERS

This appendix contains the macro parameters used for customizing the ACVC. The meaning and purpose of these parameters are explained in [UG89]. The parameter values are presented in two tables. The first table lists the values that are defined in terms of the maximum input-line length, which is the value for $MAX_IN_LEN--also listed here. These values are expressed here as Ada string aggregates, where "V" represents the maximum input-line length.

Macro Parameter	Macro Value
$BIG_ID1	(1..V-1 => 'A', V => '1')
$BIG_ID2	(1..V-1 => 'A', V => '2')
$BIG_ID3	(1..V/2 => 'A') & '3' & (1..V-1-V/2 => 'A')
$BIG_ID4	(1..V/2 => 'A') & '4' & (1..V-1-V/2 => 'A')
$BIG_INT_LIT	(1..V-3 => '0') & "298"
$BIG_REAL_LIT	(1..V-5 => '0') & "690.0"
$BIG_STRING1	'"' & (1..V/2 => 'A') & '"'
$BIG_STRING2	'"' & (1..V-1-V/2 => 'A') & '1' & '"'
$BLANKS	(1..V-20 => ' ')
$MAX_LEN_INT_BASED_LITERAL	"2:" & (1..V-5 => '0') & "11:"
$MAX_LEN_REAL_BASED_LITERAL	"16:" & (1..V-7 => '0') & "F.E:"
$MAX_STRING_LITERAL	'"' & (1..V-2 => 'A') & '"'

MACRO PARAMETERS

The following table lists all of the other macro parameters and their respective values.

Macro Parameter	Macro Value
$MAX_IN_LEN	255
$ACC_SIZE	32
$ALIGNMENT	4
$COUNT_LAST	2147483647
$DEFAULT_MEM_SIZE	2**32
$DEFAULT_STOR_UNIT	8
$DEFAULT_SYS_NAME	I80386
$DELTA_DOC	2#1.0 #E-31
$ENTRY_ADDRESS	TO_ADDRESS(16#40#)
$ENTRY_ADDRESS1	TO_ADDRESS(16#80#)
$ENTRY_ADDRESS2	TO_ADDRESS(16#100#)
$FIELD_LAST	255
$FILE_TERMINATOR	''
$FIXED_NAME	NO_SUCH_FIXED_TYPE
$FLOAT_NAME	NO_SUCH_FLOAT_TYPE
$FORM_STRING	""
$FORM_STRING2	"CANNOT_RESTRICT_FILE_CAPACITY"
$GREATER_THAN_DURATION	75000.0
$GREATER_THAN_DURATION_BASE_LAST	131073.0
$GREATER_THAN_FLOAT_BASE_LAST	1.80141E+38
$GREATER_THAN_FLOAT_SAFE_LARGE	1.0E308

$GREATER_THAN_SHORT_FLOAT_SAFE_LARGE	1.0E308
$HIGH_PRIORITY	10
$ILLEGAL_EXTERNAL_FILE_NAME1	\NODIRECTORY\FILENAME
$ILLEGAL_EXTERNAL_FILE_NAME2	THIS_FILE_NAME_IS_TOO_LONG_FOR_MY_SYSTEM
$INAPPROPRIATE_LINE_LENGTH	-1
$INAPPROPRIATE_PAGE_LENGTH	-1
$INCLUDE_PRAGMA1	PRAGMA INCLUDE ("A28006D1.TST")
$INCLUDE_PRAGMA2	PRAGMA INCLUDE ("B28006D1.TST")
$INTEGER_FIRST	-2147483648
$INTEGER_LAST	2147483647
$INTEGER_LAST_PLUS_1	2147483648
$INTERFACE_LANGUAGE	C
$LESS_THAN_DURATION	-75000.0
$LESS_THAN_DURATION_BASE_FIRST	-131073.0
$LINE_TERMINATOR	ASCII.CR & ASCII.LF
$LOW_PRIORITY	1
$MACHINE_CODE_STATEMENT	NULL;
$MACHINE_CODE_TYPE	NO_SUCH_TYPE
$MANTISSA_DOC	31
$MAX_DIGITS	15
$MAX_INT	2147483647
$MAX_INT_PLUS_1	2147483648
$MIN_INT	-2147483648

MACRO PARAMETERS

$NAME	SHORT_SHORT_INTEGER
$NAME_LIST	I80386
$NAME_SPECIFICATION1	E:\ACVC\X2120A
$NAME_SPECIFICATION2	E:\ACVC\X2120B
$NAME_SPECIFICATION3	E:\ACVC\X3119A
$NEG_BASED_INT	16#F000000E#
$NEW_MEM_SIZE	2**32
$NEW_STOR_UNIT	16
$NEW_SYS_NAME	I80386
$PAGE_TERMINATOR	ASCII.CR & ASCII.LF & ASCII.FF
$RECORD_DEFINITION	NEW INTEGER;
$RECORD_NAME	NO_SUCH_MACHINE_CODE_TYPE
$TASK_SIZE	32
$TASK_STORAGE_SIZE	1024
$TICK	1.0/18.2
$VARIABLE_ADDRESS	TO_ADDRESS(16#0020#)
$VARIABLE_ADDRESS1	TO_ADDRESS(16#0024#)
$VARIABLE_ADDRESS2	TO_ADDRESS(16#0028#)
$YOUR_PRAGMA	INTERFACE

APPENDIX B

COMPILATION SYSTEM OPTIONS

The compiler options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report.

compile Options

```
COMPILE (SOURCE     => source_name | INSTANTIATION,
         LIBRARY    => library_name,
         OPTIONS    =>
           (ANNOTATE    => character_string,
            ERRORS      => positive_integer,
            LEVEL       => PARSE | SEMANTIC | CODE | UPDATE,
            CHECKS      => ALL | STACK | NONE,
            GENERICS    => STUBS | INLINE,
            TASKING     => YES | NO,
            MEMORY      => number_of_kbytes),
         DISPLAY    =>
           (OUTPUT      => SCREEN | NONE | AUTOMATIC | file_name,
            WARNING     => YES | NO,
            TEXT        => YES | NO,
            SHOW        => BANNER | RECAP | ALL | NONE,
            DETAIL      => YES | NO,
            ASSEMBLY    => CODE | MAP | ALL | NONE),
         ALLOCATION =>
           (STACK       => positive_integer),
         IMPROVE    =>
           (CALLS       => NORMAL | INLINED,
            REDUCTION   => NONE | PARTIAL | EXTENSIVE,
            EXPRESSIONS => NONE | PARTIAL | EXTENSIVE);
         KEEP       =>
           (COPY       => YES | NO,
            DEBUG      => YES | NO,
            TREE       => YES | NO));
```

LINKER OPTIONS

The linker options of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this appendix are to linker documentation and not to this report.

```
                 compile Options
BIND (PROGRAM    => main_program_name,
      LIBRARY    => library_name,
      OPTIONS    =>
        (LEVEL          => CHECK | BIND | LINK,
         EXECUTION      => EXTENDED | PROTECTED,
         FLOAT          => AUTOMATIC | HARDWARE | SOFTWARE,
         MATHLIB        => I287 | I387,
         OBJECT         => AUTOMATIC | file_name,
         UNCALLED       => REMOVE | KEEP,
         TIMER          => NORMAL | FAST,
         SLICE          => NO | positive_integer),
      STACK      =>
        (MAIN           => positive_integer,
         TASK           => positive_integer,
         HISTORY        => YES | NO),
      HEAP       =>
        (SIZE           => positive_integer,
         INCREMENT      => positive_integer),
      INTERFACE  =>
        (DIRECTIVES     => options_for_linker,
         MODULES        => file_names,
         SEARCH         => library_names),
      DISPLAY    =>
        (OUTPUT         => SCREEN | NONE | AUTOMATIC | file_name,
         DATA           => BIND | LINK | ALL | NONE,
         WARNING        => YES | NO),
      KEEP       =>
        (DEBUG          => YES | NO,
         TUNE           => YES | NO));
```

APPENDIX C

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of this Ada implementation, as described in this Appendix, are provided by the customer. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is
   ..........
   type INTEGER is range -2147483648 .. 2147483647;
   type SHORT_INTEGER is range -32768 .. 32767;
   type SHORT_SHORT_INTEGER is range -128..127;
   type FLOAT is digits 6 range
        -2#1.11111111111111111111111#E+127 ..
         2#1.11111111111111111111111#E+127;
   type SHORT_FLOAT is digits 6 range
        -2#1.11111111111111111111111#E+127 ..
         2#1.11111111111111111111111#E+127;
   type LONG_FLOAT is digits 15 range
        2#1.1111111111111111111111111111111111111111111111111111#E1023 ..
        2#1.1111111111111111111111111111111111111111111111111111#E1023;
   type DURATION is delta 2#0.00000000000001# range
        -131072.0000 .. 131071.99994;
   ..........
end STANDARD;
```

Alsys DOS Ada Software Engineering Environment

APPENDIX F

Version 5

Printed: October 1990

Alsys reserves the right to make changes in specifications and other information contained in this publication without prior notice. Consult Alsys to determine whether such changes have been made.

TABLE OF CONTENTS

APPENDIX F

Implementation - Dependent Characteristics

This appendix summarizes the implementation-dependent characteristics of the Alsys 386 DOS Compiler (32-bit mode).

Appendix F is a required part of the *Reference Manual for the Ada Programming Language* (called the RM in this appendix).

The sections of this appendix are as follows:

1. The form, allowed places, and effect of every implementation-dependent pragma.
2. The name and the type of every implementation-dependent attribute.
3. The specification of the package SYSTEM.
4. The description of the representation clauses.
5. The conventions used for any implementation-generated name denoting implementation-dependent components.
6. The interpretation of expressions that appear in address clauses, including those for interrupts.
7. Any restrictions on unchecked conversions.
8. Any implementation-dependent characteristics of the input-output packages.
9. Characteristics of numeric types.
10. Other implementation-dependent characteristics.
11. Compiler limitations.

The name *Alsys Runtime Executive Programs* or simply *Runtime Executive* refers to the runtime library routines provided for all Ada programs. These routines implement the Ada heap, exceptions, tasking control, and other utility functions.

General systems programming notes are given in another document, the *Application Developer's Guide* (for example, parameter passing conventions needed for interface with assembly routines).

Section 1

Implementation-Dependent Pragmas

1.1 INLINE

Pragma INLINE is fully supported; however, it is not possible to inline a subprogram in a declarative part.

1.2 INTERFACE

Ada programs can interface with subprograms written in Assembler and other languages through the use of the predefined pragma INTERFACE and the implementation-defined pragma INTERFACE_NAME.

Pragma INTERFACE specifies the name of an interfaced subprogram and the name of the programming language for which parameter passing conventions will be generated. Pragma INTERFACE takes the form specified in the RM:

> **pragma INTERFACE (*language_name*, *subprogram_name*);**

where,

- *language_name* is ASSEMBLER, ADA, or C.
- *subprogram_name* is the name used within the Ada program to refer to the interfaced subprogram.

The only language names accepted by pragma INTERFACE are ASSEMBLER, ADA and C. The full implementation requirements for writing pragma INTERFACE subprograms are described in the *Application Developer's Guide*.

The language name used in the pragma INTERFACE does not have to have any relationship to the language actually used to write the interfaced subprogram. It is used only to tell the Compiler how to generate subprogram calls; that is, what kind of parameter passing techniques to use. The programmer can interface Ada programs with subroutines written in any other (compiled) language by understanding the mechanisms

used for parameter passing by the Alsys DOS Ada Compiler and the corresponding mechanisms of the chosen external language.

1.3 INTERFACE_NAME

Pragma INTERFACE_NAME associates the name of the interfaced subprogram with the external name of the interfaced subprogram. If pragma INTERFACE_NAME is not used, then the two names are assumed to be identical. This pragma takes the form:

pragma INTERFACE_NAME (*subprogram_name*, *string_literal*);

where,

- *subprogram_name* is the name used within the Ada program to refer to the interfaced subprogram.
- *string_literal* is the name by which the interfaced subprogram is referred to at link time.

The pragma INTERFACE_NAME is used to identify routines in other languages that are not named with legal Ada identifiers. Ada identifiers can only contain letters, digits, or underscores, whereas the DOS Linker allows external names to contain other characters, for example, the dollar sign ($) or commercial at sign (@). These characters can be specified in the *string_literal* argument of the pragma INTERFACE_NAME.

The pragma INTERFACE_NAME is allowed at the same places of an Ada program as the pragma INTERFACE. (Location restrictions can be found in section 13.9 of the RM.) However, the pragma INTERFACE_NAME must always occur after the pragma INTERFACE declaration for the interfaced subprogram.

The *string_literal* of the pragma INTERFACE_NAME is passed through unchanged to the DOS object file. The maximum length of the *string_literal* is 40 characters. This limit is not checked by the Compiler, but the string is truncated by the Binder to meet the Intel object module format standard.

The user must be aware however, that some tools from other vendors do not fully support the standard object file format and may restrict the length of symbols. For example, the IBM and Microsoft assemblers silently truncate symbols at 31 characters.

The *Runtime Executive* contains several external identifiers. All such identifiers begin with either the string "ADA@" or the string "ADAS@". Accordingly, names prefixed by "ADA@" or "ADAS@" should be avoided by the user.

Example

```
package SAMPLE_DATA is
    function SAMPLE_DEVICE (X: INTEGER) return INTEGER;
    function PROCESS_SAMPLE (X: INTEGER) return INTEGER;
private
    pragma INTERFACE (ASSEMBLER, SAMPLE_DEVICE);
    pragma INTERFACE (ADA, PROCESS_SAMPLE);
    pragma INTERFACE_NAME (SAMPLE_DEVICE, "DEVIO$GET_SAMPLE");
end SAMPLE_DATA;
```

1.4 INDENT

Pragma INDENT is only used with *AdaReformat*. *AdaReformat* is the Alsys reformatter which offers the functionalities of a pretty-printer in an Ada environment.

The pragma is placed in the source file and interpreted by the Reformatter. The line

```
pragma INDENT(OFF);
```

causes *AdaReformat* not to modify the source lines after this pragma, while

```
pragma INDENT(ON);
```

causes *AdaReformat* to resume its action after this pragma.

1.5 Other Pragmas

Pragmas IMPROVE and PACK are discussed in detail in the section on representation clauses and records (Chapter 4).

Pragma PRIORITY is accepted with the range of priorities running from 1 to 10 (see the definition of the predefined package SYSTEM in Section 3). Undefined priority (no pragma PRIORITY) is treated as though it were less than any defined priority value.

In addition to pragma SUPPRESS, it is possible to suppress all checks in a given compilation by the use of the Compiler option CHECKS. (See Chapter 4 of the *User's Guide*.)

Section 2

Implementation-Dependent Attributes

2.1 P'IS_ARRAY

For a prefix P that denotes any type or subtype, this attribute yields the value TRUE if P is an array type or an array subtype; otherwise, it yields the value FALSE.

2.2 E'EXCEPTION_CODE

For a prefix E that denotes an exception name, this attribute yields a value that represents the internal code of the exception. The value of this attribute is of the type INTEGER.

2.3 Attributes Used in Record Representation Clauses

In addition to the Representation Attributes of [13.7.2] and [13.7.3], the following attributes are used to form names of indirect and implicit components for use in record representation clauses, as described in Section 4.8.

'OFFSET
'RECORD_SIZE
'VARIANT_INDEX
'ARRAY_DESCRIPTOR
'RECORD_DESCRIPTION

Section 3

Specification of the package SYSTEM

The implementation does not allow the recompilation of package SYSTEM.

```
package SYSTEM is

   --   *****************************
   --   * (1) Required Definitions. *
   --   *****************************

   type NAME is (I80386);
   SYSTEM_NAME  : constant NAME := I80386;

   STORAGE_UNIT : constant := 8;
   MEMORY_SIZE  : constant := 2**32;

   -- System-Dependent Named Numbers
   MAX_INT      : constant := 2**31 - 1;
   MIN_INT      : constant := - (2**31);
   MAX_MANTISSA : constant := 31;
   FINE_DELTA   : constant := 2#1.0#E-31;
   MAX_DIGITS   : constant := 15;

   -- For the high-resolution timer, the clock resolution is
   -- 1.0 / 1024.0.
   TICK         : constant := 1.0 / 18.2;

   -- Other System-Dependent Declarations:
   subtype PRIORITY is INTEGER range 1 .. 10;
```

```
-- The type ADDRESS is, in fact, implemented as a
-- 32 bit offset.
type ADDRESS is private;
NULL_ADDRESS : constant ADDRESS;


--    *********************************
--    * (2) Operations on Addresses. *
--    *********************************

-- VALUE converts a string to an address. The syntax of the string and
-- its meaning are target dependent.
--
-- For the 80386 the syntax is:
--    "00000000"  where 00000000 is an 8 digit or less hexadecimal number
--     representing an offset either in the data segment or in
--     the code segment.
-- Example:
--    "00000008"
--
--
-- The exception CONSTRAINT_ERROR is raised if the string does not have
-- the proper syntax.

function VALUE (LEFT : in STRING) return ADDRESS;


-- IMAGE converts an address to a string.  The syntax of the returned
-- string is described in the VALUE function.

subtype ADDRESS_STRING is STRING(1..8);

function IMAGE (LEFT : in ADDRESS) return ADDRESS_STRING;


-- SAME_SEGMENT always returns TRUE for the 80386.
```

```
function SAME_SEGMENT (LEFT, RIGHT : in ADDRESS) return BOOLEAN;
```

```
-- The following routines provide support to perform address
-- computation.

type OFFSET is range 0 .. 2**31-1;

-- The exception CONSTRAINT_ERROR can be raised by "+" and "-".

ADDRESS_ERROR : exception;

function "+" (LEFT : in ADDRESS; RIGHT : in OFFSET)  return ADDRESS;
function "+" (LEFT : in OFFSET;  RIGHT : in ADDRESS) return ADDRESS;
function "-" (LEFT : in ADDRESS; RIGHT : in OFFSET)  return ADDRESS;

function "-" (LEFT : in ADDRESS; RIGHT : in ADDRESS) return OFFSET;

-- Perform an unsigned comparison on addresses.

function "<=" (LEFT, RIGHT : in ADDRESS) return BOOLEAN;
function "<"  (LEFT, RIGHT : in ADDRESS) return BOOLEAN;
function ">=" (LEFT, RIGHT : in ADDRESS) return BOOLEAN;
function ">"  (LEFT, RIGHT : in ADDRESS) return BOOLEAN;

function "mod" (LEFT : in ADDRESS; RIGHT : in POSITIVE) return NATURAL;


-- Returns the given address rounded to a specific value.

type ROUND_DIRECTION is (DOWN, UP);

function ROUND (VALUE     : in ADDRESS;
                DIRECTION : in ROUND_DIRECTION;
                MODULUS   : in POSITIVE) return ADDRESS;
```

```
   -- These routines are provided to perform READ/WRITE operation
   -- in memory.
   -- WARNING: These routines will give unexpected results if used with
   -- unconstrained types.

   generic
      type TARGET is private;
   function FETCH_FROM_ADDRESS (A : in ADDRESS) return TARGET;

   generic
      type TARGET is private;
   procedure ASSIGN_TO_ADDRESS (A : in ADDRESS; T : in TARGET);


   -- MOVE is a procedure to copy LENGTH storage unit starting at the
   -- address FROM to the address TO.  The source and destination may
   -- overlap.  OBJECT_LENGTH designates the size of an object in
   -- storage units.

   type OBJECT_LENGTH is range 0 .. 2**31 -1;

   procedure MOVE (TO     : in ADDRESS;
                   FROM   : in ADDRESS;
                   LENGTH : in OBJECT_LENGTH);
private

   ...

end SYSTEM;
```

Section 4

Support for Representation Clauses

This section explains how objects are represented and allocated by the Alsys DOS Ada compiler and how it is possible to control this using representation clauses. Applicable restrictions on representation clauses are also described.

The representation of an object is closely connected with its type. For this reason this section addresses successively the representation of enumeration, integer, floating point, fixed point, access, task, array and record types. For each class of type the representation of the corresponding objects is described.

Except in the case of array and record types, the description for each class of type is independent of the others. To understand the representation of array and record types it is necessary to understand first the representation of their components.

Apart from implementation defined pragmas, Ada provides three means to control the size of objects:

- a (predefined) pragma PACK, applicable to array types
- a record representation clause
- a size specification

For each class of types the effect of a size specification is described. Interactions among size specifications, packing and record representation clauses is described under the discussion of array and record types.

Representation clauses on derived record types or derived tasks types are not supported.

Size representation clauses on types derived from private types are not supported when the derived type is declared outside the private part of the defining package.

4.1 Enumeration Types

4.1.1 Enumeration Literal Encoding

When no enumeration representation clause applies to an enumeration type, the internal code associated with an enumeration literal is the position number of the enumeration literal. Then, for an enumeration type with n elements, the internal codes are the integers 0, 1, 2, .., n-1.

An enumeration representation clause can be provided to specify the value of each internal code as described in RM 13.3. The Alsys compiler fully implements enumeration representation clauses.

As internal codes must be machine integers the internal codes provided by an enumeration representation clause must be in the range $-2^{31} .. 2^{31}-1$.

An enumeration value is always represented by its internal code in the program generated by the compiler.

4.1.2 Enumeration Types and Object Sizes

Minimum size of an enumeration subtype

The minimum possible size of an enumeration subtype is the minimum number of bits that is necessary for representing the internal codes of the subtype values in normal binary form.

A static subtype, with a null range has a minimum size of 1. Otherwise, if m and M are the values of the internal codes associated with the first and last enumeration values of the subtype, then its minimum size L is determined as follows. For $m >= 0$, L is the smallest positive integer such that $M <= 2^L-1$. For $m < 0$, L is the smallest positive integer such that $-2^{L-1} <= m$ and $M <= 2^{L-1}-1$. For example:

```
type COLOR is (GREEN, BLACK, WHITE, RED, BLUE, YELLOW);
-- The minimum size of COLOR is 3 bits.

subtype BLACK_AND_WHITE is COLOR range BLACK .. WHITE;
-- The minimum size of BLACK_AND_WHITE is 2 bits.
```

```
subtype BLACK_OR_WHITE is BLACK_AND_WHITE range X .. X;
-- Assuming that X is not static, the minimum size of BLACK_OR_WHITE is
-- 2 bits (the same as the minimum size of its type mark BLACK_AND_WHITE).
```

Size of an enumeration subtype

When no size specification is applied to an enumeration type or first named subtype, the objects of that type or first named subtype are represented as signed machine integers. The machine provides 8, 16 and 32 bit integers, and the compiler selects automatically the smallest signed machine integer which can hold each of the internal codes of the enumeration type (or subtype). The size of the enumeration type and of any of its subtypes is thus 8, 16 or 32 bits.

When a size specification is applied to an enumeration type, this enumeration type and each of its subtypes has the size specified by the length clause. The same rule applies to a first named subtype. The size specification must of course specify a value greater than or equal to the minimum size of the type or subtype to which it applies:

```
type EXTENDED is
    (-- The usual ASCII character set.
    NUL, SOH, STX, ETX, EOT, ENQ, ACK, BEL,
    ...
    'x', 'y', 'z', '{', '|', '}', '~', DEL,

    -- Extended characters
    C_CEDILLA_CAP, U_UMLAUT, E_ACUTE, ...);

for EXTENDED'SIZE use 8;
-- The size of type EXTENDED will be one byte. Its objects will be represented
-- as unsigned 8 bit integers.
```

The Alsys compiler fully implements size specifications. Nevertheless, as enumeration values are coded using integers, the specified length cannot be greater than 32 bits.

Size of the objects of an enumeration subtype

Provided its size is not constrained by a record component clause or a pragma PACK, an object of an enumeration subtype has the same size as its subtype.

4.2 Integer Types

There are three predefined integer types in the Alsys implementation for I80386 machines:

type SHORT_SHORT_INTEGER **is range** -2**07 .. 2**07-1;

type SHORT_INTEGER **is range** -2**15 .. 2**15-1;

type INTEGER **is range** -2**31 .. 2**31-1;

4.2.1 Integer Type Representation

An integer type declared by a declaration of the form:

type T **is range** L .. R;

is implicitly derived from a predefined integer type. The compiler automatically selects the predefined integer type whose range is the smallest that contains the values L to R inclusive.

Binary code is used to represent integer values. Negative numbers are represented using two's complement.

4.2.2 Integer Type and Object Size

Minimum size of an integer subtype

The minimum possible size of an integer subtype is the minimum number of bits that is necessary for representing the internal codes of the subtype values in normal binary form.

For a static subtype, if it has a null range its minimum size is 1. Otherwise, if m and M are the lower and upper bounds of the subtype, then its minimum size L is determined as follows. For $m >= 0$, L is the smallest positive integer such that $M <= 2^{L-1}$. For $m < 0$, L is the smallest positive integer that $-2^{L-1} <= m$ and $M <= 2^{L-1}-1$. For example:

subtype S **is** INTEGER **range** 0 .. 7;
-- The minimum size of S is 3 bits.

```
subtype D is S range X .. Y;
-- Assuming that X and Y are not static, the minimum size of
-- D is 3 bits (the same as the minimum size of its type mark S).
```

Size of an integer subtype

The sizes of the predefined integer types SHORT_SHORT_INTEGER, SHORT_INTEGER and INTEGER are respectively 8, 16 and 32 bits.

When no size specification is applied to an integer type or to its first named subtype (if any), its size and the size of any of its subtypes is the size of the predefined type from which it derives, directly or indirectly. For example:

```
type S is range 80 .. 100;
-- S is derived from SHORT_SHORT_INTEGER, its size is
-- 8 bits.

type J is range 0 .. 255;
-- J is derived from SHORT_INTEGER, its size is 16 bits.

type N is new J range 80 .. 100;
-- N is indirectly derived from SHORT_INTEGER, its size is
-- 16 bits.
```

When a size specification is applied to an integer type, this integer type and each of its subtypes has the size specified by the length clause. The same rule applies to a first named subtype. The size specification must of course specify a value greater than or equal to the minimum size of the type or subtype to which it applies:

```
type S is range 80 .. 100;
for S'SIZE use 32;
-- S is derived from SHORT_SHORT_INTEGER, but its size is
-- 32 bits because of the size specification.

type J is range 0 .. 255;
for J'SIZE use 8;
-- J is derived from SHORT_INTEGER, but its size is 8 bits
-- because of the size specification.
```

```
type N is new J range 80 .. 100;
-- N is indirectly derived from SHORT_INTEGER, but its
-- size is 8 bits because N inherits the size specification
-- of J.
```

Size of the objects of an integer subtype

Provided its size is not constrained by a record component clause or a pragma PACK, an object of an integer subtype has the same size as its subtype.

4.3 Floating Point Types

There are two predefined floating point types in the Alsys implementation for I80386 machines:

```
type SHORT_FLOAT is
    digits 6 range -(2.0 - 2.0**(-23))*2.0**127 .. (2.0 - 2.0**(-23))*2.0**127;

type FLOAT is
    digits 6 range -(2.0 - 2.0**(-23))*2.0**127 .. (2.0 - 2.0**(-23))*2.0**127;

type LONG_FLOAT is
    digits 15 range -(2.0 - 2.0**(-51))*2.0**1023 .. (2.0 - 2.0**(-51))*2.0**1023;
```

Note that SHORT_FLOAT has the same range as FLOAT.

4.3.1 Floating Point Type Representation

A floating point type declared by a declaration of the form:

```
type T is digits D [range L .. R];
```

is implicitly derived from a predefined floating point type. The compiler automatically selects the smallest predefined floating point type whose number of digits is greater than or equal to D and which contains the values L to R inclusive.

In the program generated by the compiler, floating point values are represented using the IEEE standard formats for single and double floats.

The values of the predefined types SHORT_FLOAT and FLOAT are represented using the single float format. The values of the predefined type LONG_FLOAT are represented using the double float format. The values of any other floating point type are represented in the same way as the values of the predefined type from which it derives, directly or indirectly.

4.3.2 Floating Point Type and Object Size

The minimum possible size of a floating point subtype is 32 bits if its base type is SHORT_FLOAT or FLOAT or a type derived from SHORT_FLOAT or FLOAT; it is 64 bits if its base type is LONG_FLOAT or a type derived from LONG_FLOAT.

The sizes of the predefined floating point types SHORT_FLOAT and FLOAT is 32 bits and LONG_FLOAT is 64 bits.

The size of a floating point type and the size of any of its subtypes is the size of the predefined type from which it derives directly or indirectly.

The only size that can be specified for a floating point type or first named subtype using a size specification is its usual size (32 or 64 bits).

An object of a floating point subtype has the same size as its subtype.

4.4 Fixed Point Types

4.4.1 Fixed Point Type Representation

If no specification of small applies to a fixed point type, then the value of small is determined by the value of delta as defined by RM 3.5.9.

A specification of small can be used to impose a value of small. The value of small is required to be a power of two.

To implement fixed point types, the Alsys compiler for I80386 machines uses a set of anonymous predefined types of the form:

```
type SHORT_FIXED is delta D range (-2.0**7-1)*S .. 2.0**7*S;
for SHORT_FIXED'SMALL use S;
```

```
type FIXED is delta D range (-2.0**15-1)*S .. 2.0**15*S;
for FIXED'SMALL use S;

type LONG_FIXED is delta D range (-2.0**31-1)*S .. 2.0**31*S;
for LONG_FIXED'SMALL use S;
```

where D is any real value and S any power of two less than or equal to D.

A fixed point type declared by a declaration of the form:

```
type T is delta D range L .. R;
```

possibly with a small specification:

```
for T'SMALL use S;
```

is implicitly derived from a predefined fixed point type. The compiler automatically selects the predefined fixed point type whose small and delta are the same as the small and delta of T and whose range is the shortest that includes the values L to R inclusive.

In the program generated by the compiler, a safe value V of a fixed point subtype F is represented as the integer:

```
V / F'BASE'SMALL
```

4.4.2 Fixed Point Type and Object Size

Minimum size of a fixed point subtype

The minimum possible size of a fixed point subtype is the minimum number of binary digits that is necessary for representing the values of the range of the subtype using the small of the base type.

For a static subtype, if it has a null range its minimum size is 1. Otherwise, s and S being the bounds of the subtype, if i and I are the integer representations of m and M, the smallest and the greatest model numbers of the base type such that s < m and M < S, then the minimum size L is determined as follows. For i >= 0, L is the smallest positive integer such that $I <= 2^{L-1}$. For i < 0, L is the smallest positive integer such that $-2^{L-1} <= i$ and $I <= 2^{L-1}-1$.

```
type F is delta 2.0 range 0.0 .. 500.0;
-- The minimum size of F is 8 bits.
```

```
subtype S is F delta 16.0 range 0.0 .. 250.0;
-- The minimum size of S is 7 bits.

subtype D is S range X .. Y;
-- Assuming that X and Y are not static, the minimum size  of D is 7 bits
-- (the same as the minimum size of its type mark S).
```

Size of a fixed point subtype

The sizes of the predefined fixed point types SHORT_FIXED, FIXED and LONG_FIXED are respectively 8, 16 and 32 bits.

When no size specification is applied to a fixed point type or to its first named subtype, its size and the size of any of its subtypes is the size of the predefined type from which it derives directly or indirectly. For example:

```
type S is delta 0.01 range 0.8 .. 1.0;
-- S is derived from an 8 bit predefined fixed type, its size is 8 bits.

type F is delta 0.01 range 0.0 .. 2.0;
-- F is derived from a 16 bit predefined fixed type, its size is 16 bits.

type N is new F range 0.8 .. 1.0;
-- N is indirectly derived from a 16 bit predefined fixed type, its size is 16 bits.
```

When a size specification is applied to a fixed point type, this fixed point type and each of its subtypes has the size specified by the length clause. The same rule applies to a first named subtype. The size specification must of course specify a value greater than or equal to the minimum size of the type or subtype to which it applies:

```
type S is delta 0.01 range 0.8 .. 1.0;
for S'SIZE use 32;
-- S is derived from an 8 bit predefined fixed type, but its size is 32 bits
-- because of the size specification.

type F is delta 0.01 range 0.0 .. 2.0;
for F'SIZE use 8;
-- F is derived from a 16 bit predefined fixed type, but its size is 8 bits
-- because of the size specification.
```

type N **is new** F **range** 0.8 .. 1.0;
-- N is indirectly derived from a 16 bit predefined fixed type, but its size is
-- 8 bits because N inherits the size specification of F.

The Alsys compiler fully implements size specifications. Nevertheless, as fixed point objects are represented using machine integers, the specified length cannot be greater than 32 bits.

Size of the objects of a fixed point subtype

Provided its size is not constrained by a record component clause or a pragma PACK, an object of a fixed point type has the same size as its subtype.

4.5 Access Types and Collections

Access Types and Objects of Access Types

The only size that can be specified for an access type using a size specification is its usual size (32 bits).

An object of an access subtype has the same size as its subtype, thus an object of an access subtype is always 32 bits long.

Collection Size

As described in RM 13.2, a specification of collection size can be provided in order to reserve storage space for the collection of an access type.

When no STORAGE_SIZE specification applies to an access type, no storage space is reserved for its collection, and the value of the attribute STORAGE_SIZE is then 0.

The maximum size is limited by the amount of memory available.

4.6 Task Types

Storage for a task activation

As described in RM 13.2, a length clause can be used to specify the storage space (that is, the stack size) for the activation of each of the tasks of a given type. Alsys also allows the task stack size, for all tasks, to be established using a Binder option. If a length clause is given for a task type, the value indicated at bind time is ignored for this task type, and the length clause is obeyed. When no length clause is used to specify the storage space to be reserved for a task activation, the storage space indicated at bind time is used for this activation.

A length clause may not be applied to a derived task type. The same storage space is reserved for the activation of a task of a derived type as for the activation of a task of the parent type.

The minimum size of a task subtype is 32 bits.

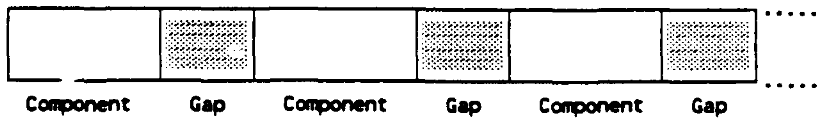
A size specification has no effect on a task type. The only size that can be specified using such a length clause is its usual size (32 bits).

An object of a task subtype has the same size as its subtype. Thus an object of a task subtype is always 32 bits long.

4.7 Array Types

Each array is allocated in a contiguous area of storage units. All the components have the same size. A gap may exist between two consecutive components (and after the last one). All the gaps have the same size.

4.7.1 Array Layout and Structure and Pragma PACK



If pragma PACK is not specified for an array, the size of the components is the size of the subtype of the components:

```
type A is array (1 .. 8) of BOOLEAN;
-- The size of the components of A is the size of the type BOOLEAN: 8 bits.

type DECIMAL_DIGIT is range 0 .. 9;
for DECIMAL_DIGIT'SIZE use 4;
type BINARY_CODED_DECIMAL is
      array (INTEGER range < >) of DECIMAL_DIGIT;
-- The size of the type DECIMAL_DIGIT is 4 bits. Thus in an array of
-- type BINARY_CODED_DECIMAL each component will be represented on
-- 4 bits as in the usual BCD representation.
```

If pragma PACK is specified for an array and its components are neither records nor arrays, the size of the components is the minimum size of the subtype of the components:

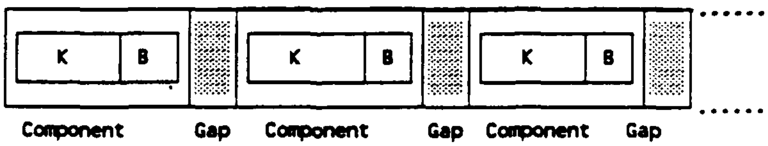
```
type A is array (1 .. 8) of BOOLEAN;
pragma PACK(A);
-- The size of the components of A is the minimum size of the type BOOLEAN:
-- 1 bit.

type DECIMAL_DIGIT is range 0 .. 9;
for DECIMAL_DIGIT'SIZE use 32;
type BINARY_CODED_DECIMAL is
      array (INTEGER range < >) of DECIMAL_DIGIT;
pragma PACK(BINARY_CODED_DECIMAL);
-- The size of the type DECIMAL_DIGIT is 32 bits, but, as
-- BINARY_CODED_DECIMAL is packed, each component of an array of this
-- type will be represented on 4 bits as in the usual BCD representation.
```

Packing the array has no effect on the size of the components when the components are records or arrays, since records and arrays may be assigned addresses consistent with the alignment of their subtypes.

Gaps

If the components are records or arrays, no size specification applies to the subtype of the components and the array is not packed, then the compiler may choose a representation with a gap after each component; the aim of the insertion of such gaps is to optimize access to the array components and to their subcomponents. The size of the gap is chosen so that the relative displacement of consecutive components is a multiple of the alignment of the subtype of the components. This strategy allows each component and subcomponent to have an address consistent with the alignment of its subtype:

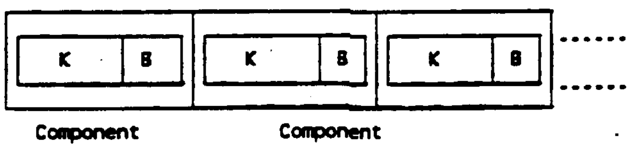
```
type R is
      record
            K : SHORT_INTEGER;
            B : BOOLEAN;
      end record;
for R use
      record
            K at 0 range 0 .. 31;
            B at 4 range 0 .. 0;
      end record;
      -- Record type R is byte aligned. Its size is 33 bits.

type A is array (1 .. 10) of R;
-- A gap of 7 bits is inserted after each component in order to respect the
-- alignment of type R. The size of an array of type A will be 400 bits.
```



Array of type A: each subcomponent K has an even offset.

If a size specification applies to the subtype of the components or if the array is packed, no gaps are inserted:

```
type R is
    record
        K : SHORT_INTEGER;
        B : BOOLEAN;
    end record;

type A is array (1 .. 10) of R;
pragma PACK(A);
-- There is no gap in an array of type A because A is packed.
-- The size of an object of type A will be 330 bits.

type NR is new R;
for NR'SIZE use 24;

type B is array (1 .. 10) of NR;
-- There is no gap in an array of type B because
-- NR has a size specification.
-- The size of an object of type B will be 240 bits.
```



Array of type A or B

4.7.2 Array Subtype and Object Size

Size of an array subtype

The size of an array subtype is obtained by multiplying the number of its components by the sum of the size of the components and the size of the gaps (if any). If the subtype is unconstrained, the maximum number of components is considered.

The size of an array subtype cannot be computed at compile time

- if it has non-static constraints or is an unconstrained array type with non-static index subtypes (because the number of components can then only be determined at run time).
- if the components are records or arrays and their constraints or the constraints of their subcomponents (if any) are not static (because the size of the components and the size of the gaps can then only be determined at run time).

As has been indicated above, the effect of a pragma PACK on an array type is to suppress the gaps. The consequence of packing an array type is thus to reduce its size.

If the components of an array are records or arrays and their constraints or the constraints of their subcomponents (if any) are not static, the compiler ignores any pragma PACK applied to the array type but issues a warning message. Apart from this limitation, array packing is fully implemented by the Alsys compiler.

A size specification applied to an array type or first named subtype has no effect. The only size that can be specified using such a length clause is its usual size. Nevertheless, such a length clause can be useful to verify that the layout of an array is as expected by the application.

Size of the objects of an array subtype

The size of an object of an array subtype is always equal to the size of the subtype of the object.

4.8 Record Types

4.8.1 Basic Record Structure

Layout of a record

Each record is allocated in a contiguous area of storage units. The size of a record component depends on its type.

The positions and the sizes of the components of a record type object can be controlled using a record representation clause as described in RM 13.4. In the Alsys

implementation for I80386 machines there is no restriction on the position that can be specified for a component of a record. If a component is not a record or an array, its size can be any size from the minimum size to the size of its subtype. If a component is a record or an array, its size must be the size of its subtype.

```
type DEVICE_INFO_RECORD is
  record
    BIT15   : BOOLEAN;   -- Bit 15 (reserved)
    CTRL    : BOOLEAN;   -- Bit 14 (true if control strings processed)
    NETWORK : BOOLEAN;   -- Bit 13 (true if device is on network)
    BIT12   : BOOLEAN;   -- Bit 12 (reserved)
    BIT11   : BOOLEAN;   -- Bit 11 (reserved)
    BIT10   : BOOLEAN;   -- Bit 10 (reserved)
    BIT9    : BOOLEAN;   -- Bit 9 (reserved)
    BIT8    : BOOLEAN;   -- Bit 8 (reserved)
    ISDEV   : BOOLEAN;   -- Bit 7 (true if device, false if disk file)
    EOF     : BOOLEAN;   -- Bit 6 (true if at end of file)
    BINARY  : BOOLEAN;   -- Bit 5 (true if binary (raw) mode)
    BIT4    : BOOLEAN;   -- Bit 4 (reserved)
    ISCLK   : BOOLEAN;   -- Bit 3 (true if clock device)
    ISNUL   : BOOLEAN;   -- Bit 2 (true if NUL device)
    ISCOT   : BOOLEAN;   -- Bit 1 (true if console output device)
    ISCIN   : BOOLEAN;   -- Bit 0 (true if console input device)
  end record;


for DEVICE_INFO_RECORD use
  record
    BIT15    at 1 range 7 .. 7;   -- Bit 15
    CTRL     at 1 range 6 .. 6;   -- Bit 14
    NETWORK  at 1 range 5 .. 5;   -- Bit 13
    BIT12    at 1 range 4 .. 4;   -- Bit 12
    BIT11    at 1 range 3 .. 3;   -- Bit 11
    BIT10    at 1 range 2 .. 2;   -- Bit 10
    BIT9     at 1 range 1 .. 1;   -- Bit 9
    BIT8     at 1 range 0 .. 0;   -- Bit 8
```

```
    ISDEV     at 0 range 7 .. 7;    -- Bit 7
    EOF       at 0 range 6 .. 6;    -- Bit 6
    BINARY    at 0 range 5 .. 5;    -- Bit 5
    BIT4      at 0 range 4 .. 4;    -- Bit 4
    ISCLK     at 0 range 3 .. 3;    -- Bit 3
    ISNUL     at 0 range 2 .. 2;    -- Bit 2
    ISCOT     at 0 range 1 .. 1;    -- Bit 1
    ISCIN     at 0 range 0 .. 0;    -- Bit 0
end record;
```

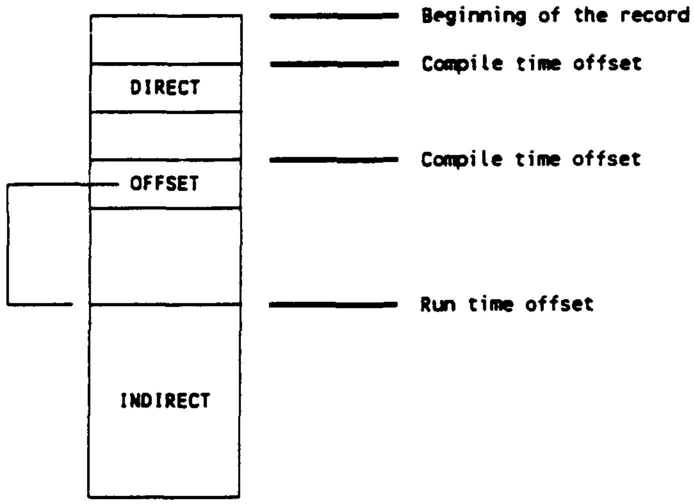
Pragma PACK has no effect on records. It is unnecessary because record representation clauses provide full control over record layout.

A record representation clause need not specify the position and the size for every component. If no component clause applies to a component of a record, its size is the size of its subtype.

4.8.2 Indirect Components

'OFFSET

If the offset of a component cannot be computed at compile time, this offset is stored in the record objects at run time and used to access the component. Such a component is said to be indirect while other components are said to be direct:



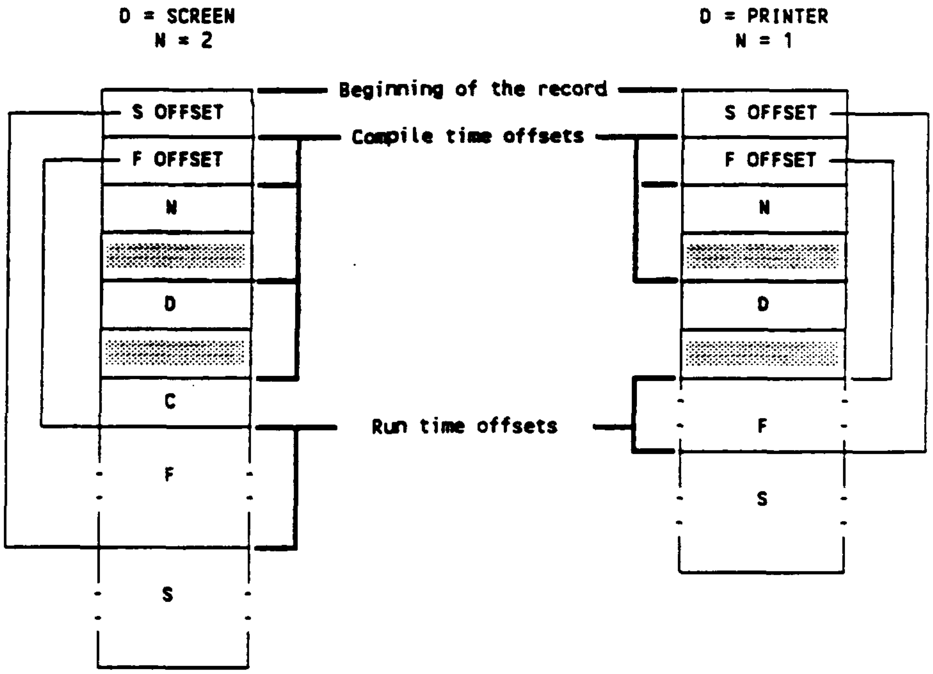
A direct and an indirect component

If a record component is a record or an array, the size of its subtype may be evaluated at run time and may even depend on the discriminants of the record. We will call these components dynamic components:

```
type DEVICE is (SCREEN, PRINTER);

type COLOR is (GREEN, RED, BLUE);

type SERIES is array (POSITIVE range < >) of INTEGER;

type GRAPH (L : NATURAL) is
    record
        X : SERIES(1 .. L); -- The size of X depends on L
        Y : SERIES(1 .. L); -- The size of Y depends on L
    end record;

Q : POSITIVE;
```

```
type PICTURE (N : NATURAL; D : DEVICE) is
   record
      F : GRAPH(N); -- The size of F depends on N
      S : GRAPH(Q); -- The size of S depends on Q
      case D is
       when SCREEN =>
         C : COLOR;
       when PRINTER =>
         null;
      end case;
   end record;
```

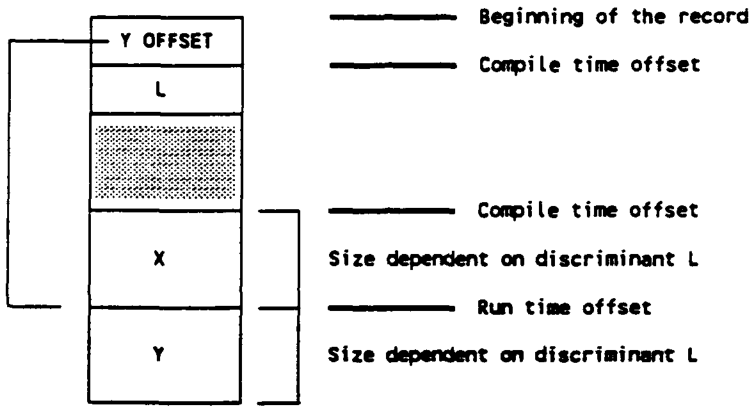
Any component placed after a dynamic component has an offset which cannot be evaluated at compile time and is thus indirect. In order to minimize the number of indirect components, the compiler groups the dynamic components together and places them at the end of the record:



The record type PICTURE: F and S are placed at the end of the record

Note that Ada does not allow representation clauses for record components with non-static bounds [RM 13.4.7], so the compiler's grouping of dynamic components does not conflict with the use of representation clauses.

Because of this approach, the only indirect components are dynamic components. But not all dynamic components are necessarily indirect: if there are dynamic components in a component list which is not followed by a variant part, then exactly one dynamic component of this list is a direct component because its offset can be computed at compilation time (the only dynamic components that are direct components are in this situation):



The record type GRAPH: the dynamic component X is a direct component.

The offset of an indirect component is always expressed in storage units.

The space reserved for the offset of an indirect component must be large enough to store the size of any value of the record type (the maximum potential offset). The compiler evaluates an upper bound MS of this size and treats an offset as a component having an anonymous integer type whose range is 0 .. MS.

If C is the name of an indirect component, then the offset of this component can be denoted in a component clause by the implementation generated name C'OFFSET.

4.8.3 Implicit Components

In some circumstances, access to an object of a record type or to its components involves computing information which only depends on the discriminant values. To avoid recomputation (which would degrade performance) the compiler stores this information in the record objects, updates it when the values of the discriminants are modified and uses it when the objects or its components are accessed. This information is stored in special components called implicit components.

An implicit component may contain information which is used when the record object or several of its components are accessed. In this case the component will be included in any record object (the implicit component is considered to be declared before any variant part in the record type declaration). There can be two components of this kind; one is called RECORD_SIZE and the other VARIANT_INDEX.

On the other hand an implicit component may be used to access a given record component. In that case the implicit component exists whenever the record component exists (the implicit component is considered to be declared at the same place as the record component). Components of this kind are called ARRAY_DESCRIPTORs or RECORD_DESCRIPTORs.

'RECORD_SIZE

This implicit component is created by the compiler when the record type has a variant part and its discriminants are defaulted. It contains the size of the storage space necessary to store the current value of the record object (note that the storage effectively allocated for the record object may be more than this).

The value of a RECORD_SIZE component may denote a number of bits or a number of storage units. In general it denotes a number of storage units, but if any component clause specifies that a component of the record type has an offset or a size which cannot be expressed using storage units, then the value designates a number of bits.

The implicit component RECORD_SIZE must be large enough to store the maximum size of any value of the record type. The compiler evaluates an upper bound MS of this size and then considers the implicit component as having an anonymous integer type whose range is 0 .. MS.

If R is the name of the record type, this implicit component can be denoted in a component clause by the implementation generated name R'RECORD_SIZE. This allows user control over the position of the implicit component in the record.

VARIANT_INDEX

This implicit component is created by the compiler when the record type has a variant part. It indicates the set of components that are present in a record value. It is used when a discriminant check is to be done.

Component lists in variant parts that themselves do not contain a variant part are numbered. These numbers are the possible values of the implicit component VARIANT_INDEX.

```
type VEHICLE is (AIRCRAFT, ROCKET, BOAT, CAR);

type DESCRIPTION (KIND : VEHICLE := CAR) is
   record
      SPEED : INTEGER;
      case KIND is
        when AIRCRAFT | CAR =>
          WHEELS : INTEGER;
          case KIND is
            when AIRCRAFT =>         -- 1
              WINGSPAN : INTEGER;
            when others =>     -- 2
              null;
          end case;
        when BOAT => -- 3
          STEAM  : BOOLEAN;
        when ROCKET =>       -- 4
          STAGES : INTEGER;
      end case;
end record;
```

The value of the variant index indicates the set of components that are present in a record value:

Variant Index	Set
1	(KIND, SPEED, WHEELS, WINGSPAN)
2	(KIND, SPEED, WHEELS)
3	(KIND, SPEED, STEAM)
4	(KIND, SPEED, STAGES)

A comparison between the variant index of a record value and the bounds of an interval is enough to check that a given component is present in the value:

Component	Interval
KIND	--
SPEED	--
WHEELS	1 .. 2
WINGSPAN	1 .. 1
STEAM	3 .. 3
STAGES	4 .. 4

The implicit component VARIANT_INDEX must be large enough to store the number V of component lists that don't contain variant parts. The compiler treats this implicit component as having an anonymous integer type whose range is 1 .. V.

If R is the name of the record type, this implicit component can be denoted in a component clause by the implementation generated name R'VARIANT_INDEX. This allows user control over the position of the implicit component in the record.

'ARRAY_DESCRIPTOR

An implicit component of this kind is associated by the compiler with each record component whose subtype is an anonymous array subtype that depends on a discriminant of the record. It contains information about the component subtype.

The structure of an implicit component of kind ARRAY_DESCRIPTOR is not described in this documentation. Nevertheless, if a programmer is interested in specifying the location of a component of this kind using a component clause, size of the component may be obtained using the ASSEMBLY parameter in the COMPILE command.

The compiler treats an implicit component of the kind ARRAY_DESCRIPTOR as having an anonymous array type. If C is the name of the record component whose subtype is described by the array descriptor, then this implicit component can be denoted in a component clause by the implementation generated name C'ARRAY_DESCRIPTOR. This allows user control over the position of the implicit component in the record.

'RECORD_DESCRIPTOR

An implicit component of this kind is associated by the compiler with each record component whose subtype is an anonymous record subtype that depends on a discriminant of the record. It contains information about the component subtype.

The structure of an implicit component of kind RECORD_DESCRIPTOR is not described in this documentation. Nevertheless, if a programmer is interested in specifying the location of a component of this kind using a component clause, the size of the component may be obtained using the ASSEMBLY parameter in the COMPILE command.

The compiler treats an implicit component of the kind RECORD_DESCRIPTOR as having an anonymous array type. If C is the name of the record component whose subtype is described by the record descriptor, then this implicit component can be denoted in a component clause by the implementation generated name C'RECORD_DESCRIPTOR. This allows user control over the position of the implicit component in the record.

Suppression of Implicit Components

The Alsys implementation provides the capability of suppressing the implicit components RECORD_SIZE and/orVARIANT_INDEX from a record type. This can be done using an implementation defined pragma called IMPROVE. The syntax of this pragma is as follows:

pragma IMPROVE (TIME | SPACE , [ON =>] simple_name);

The first argument specifies whether TIME or SPACE is the primary criterion for the choice of the representation of the record type that is denoted by the second argument.

If TIME is specified, the compiler inserts implicit components as described above. If on the other hand SPACE is specified, the compiler only inserts a VARIANT_INDEX or a RECORD_SIZE component if this component appears in a record representation clause that applies to the record type. A record representation clause can thus be used to keep one implicit component while suppressing the other.

A pragma IMPROVE that applies to a given record type can occur anywhere that a representation clause is allowed for this type.

4.8.4 Size of Record Types and Objects

Size of a record subtype

Unless a component clause specifies that a component of a record type has an offset or a size which cannot be expressed using storage units, the size of a record subtype is rounded up to a whole number of storage units.

The size of a constrained record subtype is obtained by adding the sizes of its components and the sizes of its gaps (if any). This size is not computed at compile time

- when the record subtype has non-static constraints,
- when a component is an array or a record and its size is not computed at compile time.

The size of an unconstrained record subtype is obtained by adding the sizes of the components and the sizes of the gaps (if any) of its largest variant. If the size of a component or of a gap cannot be evaluated exactly at compile time an upper bound of this size is used by the compiler to compute the subtype size.

A size specification applied to a record type or first named subtype has no effect. The only size that can be specified using such a length clause is its usual size. Nevertheless, such a length clause can be useful to verify that the layout of a record is as expected by the application.

Size of an object of a record subtype

An object of a constrained record subtype has the same size as its subtype.

An object of an unconstrained record subtype has the same size as its subtype if this size is less than or equal to 8K bytes. If the size of the subtype is greater than this, the object has the size necessary to store its current value; storage space is allocated and released as the discriminants of the record change.

Section 5

Conventions for Implementation-Generated Names

The following forms of implementation-generated names [13.4(8)] are used to denote implementation-dependent record components, as described in Section 4.8 in the sections on indirect and implicit components:

C'OFFSET
R'RECORD_SIZE
R'VARIANT_INDEX
R'ARRAY_DESCRIPTORs
R'RECORD_DESCRIPTORs

where C is the name of a record component and R the name of a record type.

The following predefined packages are reserved to Alsys and cannot be recompiled:

ALSYS_BASIC_IO
ALSYS_ADA_RUNTIME
ALSYS_BASIC_DIRECT_IO
ALSYS_BASIC_SEQUENTIAL_IO

Section 6

Address Clauses

6.1 Address Clauses for Objects

An address clause can be used to specify an address for an object as described in RM 13.5. hen such a clause applies to an object, the compiler does not allocate storage for the object. The program accesses the object using the address specified in the clause. It is the responsibility of the user therefore to make sure that a valid allocation of storage has been done at the specified address.

An address clause is not allowed for task objects, for unconstrained records whose size is greater than 8k bytes or for a constant.

There are a number of ways to compose a legal address expression for use in an address clause. The most direct ways are:

- For the case where the memory is defined in Ada as another object, use the 'ADDRESS attribute to obtain the argument for the address clause for the second object.
- For the case where the desired location is memory defined in assembly or another non-Ada language (is relocatable), an interfaced routine may be used to obtain the appropriate address from referencing information known to the other language.
- For the case where an address of an object is known by its physical address, it must be mapped to the PharLap data segment before it can be accessed via an address clause. The reason being that SYSTEM.ADDRESS is a 32 bit offset in the standard PharLap data segment.

Three Ada callable assembler routines are included in the Alsys Runtime to perform physical address mapping. These routines are ADA@MAP_PHYSICAL, ADA@MAP_PHYS_ADDR, and ADA@GET_PHYS_ADDR. ADA@MAP_PHYSICAL maps physical pages into the Ada address space. ADA@MAP_PHYS_ADDR maps pages that contain specified physical address and size into the Ada address space. ADA@GET_PHYS_ADDR returns the physical address that corresponds to a given Ada

SYSTEM.ADDRESS. To call these routines from an Ada program, use the following specifications:

```
function MAP_PHYSICAL (PHYSICAL_ADDR : INTEGER;  -- physical address
                       PAGE_SIZE     : INTEGER)  -- size in pages
   return SYSTEM.ADDRESS;  -- virtual address
pragma INTERFACE      (ASSEMBLER, MAP_PHYSICAL);
pragma INTERFACE_NAME (MAP_PHYSICAL, "ADA2MAP_PHYSICAL");
```

where:

PHYSICAL_ADDR is the physical address of memory pages to map and must be a multiple of 4K.

PAGE_SIZE is the number of physical 4K byte memory pages to map.

```
function MAP_PHYS_ADDR (PHYSICAL_ADDR : INTEGER;  -- physical address
                        MEMORY_SIZE   : INTEGER)  -- size in bytes
   return SYSTEM.ADDRESS;  -- virtual address
pragma INTERFACE      (ASSEMBLER, MAP_PHYS_ADDR);
pragma INTERFACE_NAME (MAP_PHYSICAL, "ADA2MAP_PHYS_ADDR");
```

where:

PHYSICAL_ADDR is the physical address of memory to map.

MEMORY_SIZE is the number of bytes of physical memory to map.

Note: The entire page or pages that contain the physical address are mapped. If there are other objects on the same page, there is no need to call MAP_PHYS_ADDR for those objects as that page is already mapped. Use the address arithmetic routines in package SYSTEM to create the virtual addresses for those other objects.

```
function GET_PHYS_ADDR (SYSTEM_ADDR : SYSTEM.ADDRESS) -- virtual address
   return INTEGER;  -- physical address
pragma INTERFACE      (ASSEMBLER, GETMAP_PHYS_ADDR);
pragma INTERFACE_NAME (MAP_PHYSICAL, "ADA@GET_PHYS_ADDR");
```

where:

SYSTEM_ADDR is the Ada system address of an object.

With these specifications visible from your Ada program, the functions MAP_PHYSICAL and MAP_PHYS_ADDR can be called to convert a known physical address to a virtual address of type SYSTEM.ADDRESS, which can be used in address clauses.

For example, if there is a memory mapped device whose control register is 10 bytes long, with two 32 bit fields and one 16 bit field, located at physical address 800150 (hex). Assuming the above MAP_PHYS_ADDR function is visible, this control register can be accessed with the following declaration:

```
type CONTROL_REG_TYPE is
   record
      F1: INTEGER;
      F2: INTEGER;
      F3: SHORT_INTEGER;
   end record;

for CONTROL_REG_TYPE use
   record
      F1 at 0 range  0..31;
      F2 at 0 range 31..63;
      F3 at 0 range 64..79;
   end record;

CONTROL_REG : CONTROL_REG_TYPE;
for CONTROL_REG use at MAP_PHYS_ADDR (16#800150#, 10);
```

Note that very call to MAP_PHYSICAL or MAP_PHYS_ADDR causes a new entry to be created in the PharLap page table, even if every call is for the same physical page. Make sure that for every physical page, MAP_PHYSICAL and MAP_PHYS_ADDR is executed

only once during your program's execution. If declaration such as the one above for CONTROL_REG is placed within the declarative part of a procedure, and the procedure is called repeatedly, one page table entry will be created for CONTROL_REG on every procedure call. Thus memory is wasted and you can eventually get a STORAGE_ERROR or CONSTRAINT_ERROR if the procedure is called continuously and memory does run out. The proper place to put CONTROL_REG, for this case, is within the declarative part of a package where MAP_PHYSICAL or MAP_PHYS_ADDR will be executed only once during program execution.

6.2 Address Clauses for Program Units

Address clauses for program units are not implemented in the current version of the compiler.

6.3 Address Clauses for Interrupt Entries

Address clauses for interrupt entries are supported. (See Chapter 7 of the *Application Developer's Guide* for details.)

Section 7

Unchecked Conversions

Unchecked type conversions are described in [13.10.2]. The following restrictions apply to their use:

- Unconstrained arrays are not allowed as target types. Unconstrained record types without defaulted discriminants are not allowed as target types. Access types to unconstrained arrays are not allowed as target or source types. Notes also that UNCHECKED_CONVERSION cannot be used for an access to an unconstrained string.
- If the target type has a smaller size than the source type then the target is made of the least significant bits of the source.

If the source and the target types are each of scalar or access type or if they are both of composite type, the effect of the function is to return the operand.

In other cases the effect of unchecked conversion can be considered as a copy:

- If an unchecked conversion of a scalar or access source type to a composite target type is performed, the result is a copy of the source operand. The result has the size of the source.
- If an unchecked conversion of a composite source type to a scalar or access target type is performed, the result is a copy of the source operand. The result has the size of the target.

Section 8

Input-Output Packages

The RM defines the predefined input-output packages SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO, and describes how to use the facilities available within these packages. The RM also defines the package IO_EXCEPTIONS, which specifies the exceptions that can be raised by the predefined input-output packages.

In addition the RM outlines the package LOW_LEVEL_IO, which is concerned with low-level machine-dependent input-output, such as would possibly be used to write device drivers or access device registers. LOW_LEVEL_IO has not been implemented. The use of interfaced subprograms is recommended as an alternative.

8.1 Correspondence between External Files and DOS Files

Ada input-output is defined in terms of external files. Data is read from and written to external files. Each external file is implemented as a standard DOS file, including the use of STANDARD_INPUT and STANDARD_OUTPUT.

The name of an external file can be either

- the null string
- a DOS filename
- a DOS special file or device name (for example, CON and PRN)

If the name is a null string, the associated external file is a temporary file and will cease to exist when the program is terminated. The file will be placed in the current directory and its name will be chosen by DOS.

If the name is a DOS filename, the filename will be interpreted according to standard DOS conventions (that is, relative to the current directory). The exception NAME_ERROR is raised if the name part of the filename has more than 8 characters or if the extension part has more than 3 characters.

If an existing DOS file is specified to the CREATE procedure, the contents of the file will be deleted before writing to the file.

If a non-existing directory is specified in a file path name to CREATE, the directory will not be created, and the exception NAME_ERROR is raised.

8.2 Error Handling

DOS errors are translated into Ada exceptions, as defined in the RM by package IO_EXCEPTIONS. In particular, DEVICE_ERROR is raised in cases of drive not ready, unknown media, disk full or hardware errors on the disk (such as read or write fault).

8.3 The FORM Parameter

The form parameter is a string, formed from a list of attributes, with attributes separated by commas. The string is not case sensitive. The attributes specify:

- Buffering

    BUFFER_SIZE => *size_in_bytes*

- Appending

    APPEND => YES | <u>NO</u>

- Truncation of the name by DOS

    TRUNCATE => YES | <u>NO</u>

- DIRECT_IO on UNCONSTRAINED objects

    RECORD_SIZE => *size_in_bytes*

where:

> **BUFFER_SIZE:** Controls the size of the internal buffer. This option is not supported for DIRECT_IO. The default value is 1024. This option has no effect when used by TEXT_IO with an external file that is a character device, in which case the size of the buffer will be 0.

APPEND: If YES output is appended to the end of the existing file. If NO output overwrites the existing file. This option is not supported for DIRECT_IO. The default is NO.

TRUNCATE: If YES the file name will be automatically truncated if it is bigger than 8 characters. The default value is NO, meaning that the exception NAME_ERROR will be raised if the name is too long.

RECORD_SIZE: This option is supported only for DIRECT_IO. This attribute controls the logical record length of the external file.

- When DIRECT_IO is instantiated with an unconstrained type the user is required to specify the RECORD_SIZE attribute (otherwise USE_ERROR will be raised). The value given must be larger or equal to the largest record which is going to written. If a larger record is processed the exception USE_ERROR will be raised.
- When DIRECT_IO is instantiated with a constrained type the user is not required to specify the RECORD_SIZE but if the RECORD_SIZE is specified the only possible value would be the element size in bytes. Any other values will raise USE_ERROR.

The exception USE_ERROR is raised if the form STRING in not correct or if a non supported attribute for a given package is used.

Example:

FORM => "TRUNCATE => YES, APPEND => YES, BUFFER_SIZE => 20480"

8.4 Sequential Files

For sequential access the file is viewed as a sequence of values that are transferred in the order of their appearance (as produced by the program or run-time environment). This is sometimes called a stream file in other operating systems. Each object in a sequential file has the same binary representation as the Ada object in the executable program.

8.5 Direct Files

For direct access the file is viewed as a set of elements occupying consecutive positions in a linear order. The position of an element in a direct file is specified by its index, which is an integer of subtype POSITIVE_COUNT.

DIRECT_IO only allows input-output for constrained types. If DIRECT_IO is instantiated for an unconstrained type, all calls to CREATE or OPEN will raise USE_ERROR. Each object in a direct file will have the same binary representation as the Ada object in the executable program. All elements within the file will have the same length.

8.6 Text Files

Text files are used for the input and output of information in ASCII character form. Each text file is a sequence of characters grouped into lines, and lines are grouped into a sequence of pages.

All text file column numbers, line numbers, and page numbers are values of the subtype POSITIVE_COUNT.

Note that due to the definitions of line terminator, page terminator, and file terminator in the RM, and the method used to mark the end of file under DOS, some ASCII files do not represent well-formed TEXT_IO files.

A text file is buffered by the *Runtime Executive* unless

- it names a device (for example, CON or PRN).
- it is STANDARD_INPUT or STANDARD_OUTPUT band has not been redirected.

If not redirected, prompts written to STANDARD_OUTPUT with the procedure PUT will appear before (or when) a GET (or GET_LINE) occurs.

The functions END_OF_PAGE and END_OF_FILE always return FALSE when the file is a device, which includes the use of the file CON, and STANDARD_INPUT when it is not redirected. Programs which would like to check for end of file when the file may be a device should handle the exception END_ERROR instead, as in the following example:

Example

```
begin
    loop
        -- Display the prompt:
        TEXT_IO.PUT ("--> ");
        -- Read the next line:
        TEXT_IO.GET_LINE (COMMAND, LAST);
        -- Now do something with COMMAND (1 .. LAST)
    end loop;
exception
    when TEXT_IO.END_ERROR =>
        null;
end;
```

END_ERROR is raised for STANDARD_INPUT when ^Z (ASCII.SUB) is entered at the keyboard.

8.7 Access Protection of External Files

All DOS access protections exist when using files under DOS. If a file is open for read only access by one process it can not be opened by another process for read/write access.

8.8 The Need to Close a File Explicitly

The *Runtime Executive* will flush all buffers and close all open files when the program is terminated, either normally or through some exception.

However, the RM does not define what happens when a program terminates without closing all the opened files. Thus a program which depends on this feature of the *Runtime Executive* might have problems when ported to another system.

8.9 Limitation on the Procedure RESET

An internal file opened for input cannot be RESET for output. However, an internal file opened for output can be RESET for input, and can subsequently be RESET back to output.

8.10 Sharing of External Files and Tasking Issues

Several internal files can be associated with the same external file only if all the internal files are opened with mode IN_MODE. However, if a file is opened with mode OUT_MODE and then changed to IN_MODE with the RESET procedure, it cannot be shared.

Care should be taken when performing multiple input-output operations on an external file during tasking because the order of calls to the I/O primitives is unpredictable. For example, two strings output by TEXT_IO.PUT_LINE in two different tasks may appear in the output file with interleaved characters. Synchronization of I/O in cases such as this is the user's responsibility.

The TEXT_IO files STANDARD_INPUT and STANDARD_OUTPUT are shared by all tasks of an Ada program.

If TEXT_IO.STANDARD_INPUT is not redirected, it will not block a program on input. All tasks not waiting for input will continue running.

Section 9

Characteristics of Numeric Types

9.1 Integer Types

The ranges of values for integer types declared in package STANDARD are as follows:

SHORT_SHORT_INTEGER	-128 .. 127	-- 2**7 - 1
SHORT_INTEGER	-32768 .. 32767	-- 2**15 - 1
INTEGER	-2147483648 .. 2147483647	-- 2**31 - 1

For the packages DIRECT_IO and TEXT_IO, the range of values for types COUNT and POSITIVE_COUNT are as follows:

COUNT	0 .. 2147483647	-- 2**31 - 1
POSITIVE_COUNT	1 .. 2147483647	-- 2**31 - 1

For the package TEXT_IO, the range of values for the type FIELD is as follows:

FIELD	0 .. 255	-- 2**8 - 1

9.2 Floating Point Type Attributes

	SHORT_FLOAT and FLOAT	LONG_FLOAT
DIGITS	6	15
MANTISSA	21	51
EMAX	84	204

EPSILON	9.53674E-07	8.88178E-16
LARGE	1.93428E+25	2.57110E+61
SAFE_EMAX	125	1021
SAFE_SMALL	1.17549E-38	2.22507E-308
SAFE_LARGE	4.25353E+37	2.24712E+307
FIRST	-3.40282E+38	-1.79769E+308
LAST	3.40282E+38	1.79769E+308
MACHINE_RADIX	2	2
MACHINE_EMAX	128	1024
MACHINE_EMIN	-125	-1021
MACHINE_ROUNDS	true	true
MACHINE_OVERFLOWS	false	false
SIZE	32	64

9.3 Attributes of Type DURATION

DURATION'DELTA	2.0 ** (-14)
DURATION'SMALL	2.0 ** (-14)
DURATION'FIRST	-131_072.0
DURATION'LAST	131_072.0
DURATION'LARGE	same as DURATION'LAST

Section 10

Other Implementation-Dependent Characteristics

10.1 Use of the Floating-Point Coprocessor

Floating point coprocessor instructions are used in programs that perform arithmetic on floating point values in some fixed point operations and when the FLOAT_IO or FIXED_IO packages of TEXT_IO are used. The mantissa of a fixed point value may be obtained through a conversion to an appropriate integer type. This conversion does not use floating point operations. Object code running on the 80386 using floating point instructions can still execute without the coprocessor if the software floating point emulation is linked with the object code (see Binder option FLOAT in *User's Guide*, Section 5.2). See Appendix D of the *Application Developer's Guide* for more details.

If a program requiring floating point operation is not linked with the floating point emulator, the *Runtime Executive* will detect the absence of the floating point coprocessor by raising CONSTRAINT_ERROR.

10.2 Characteristics of the Heap

All objects created by allocators go into the heap. Also, portions of the *Runtime Executive* representation of task objects, including the task stacks, are allocated in the heap.

UNCHECKLD_DEALLOCATION is implemented for all Ada access objects except access objects to tasks. Use of UNCHECKED_DEALLOCATION on a task object will lead to unpredictable results.

All objects whose visibility is linked to a subprogram, task body, or block have their storage reclaimed at exit, whether the exit is normal or due to an exception. Effectively pragma CONTROLLED is automatically applied to all access types. Moreover, all compiler temporaries on the heap (generated by such operations as function calls returning unconstrained arrays, or many concatenations) allocated in a scope are deallocated upon leaving the scope.

Note that the programmer may force heap reclamation of temporaries associated with any statements by enclosing the statement in a begin .. end block. This is especially useful when complex concatenations or other heap-intensive operations are performed in loops, and can reduce or eliminate STORAGE_ERRORs that might otherwise occur.

The maximum size of the heap is limited only by available memory. This includes the amount of physical memory (RAM) and the amount of virtual memory (hard disk swap space).

10.3 Characteristics of Tasks

The default task stack size is 1K bytes (32K bytes for the environment task), but by using the Binder option STACK.TASK the size for all task stacks in a program may be set to a size from 1K bytes to 64K bytes.

Normal priority rules are followed for preemption, where PRIORITY values are in the range 1 .. 10. A task with *undefined* priority (no pragma PRIORITY) is considered to be lower than priority 1.

The maximum number of active tasks is restricted only by memory usage.

The accepter of a rendezvous executes the accept body code in its own stack. Rendezvous with an empty accept body (for synchronization) does not cause a context switch.

The main program waits for completion of all tasks dependent upon library packages before terminating.

Abnormal completion of an aborted task takes place immediately, except when the abnormal task is the caller of an entry that is engaged in a rendezvous, or if it is in the process of activating some tasks. Any such task becomes abnormally completed as soon as the state in question is exited.

The message

```
GLOBAL BLOCKING SITUATION DETECTED
```

is printed to STANDARD_OUTPUT when the *Runtime Executive* detects that no further progress is possible for any task in the program. The execution of the program is then abandoned.

10.4 Definition of a Main Subprogram

A library unit can be used as a main subprogram if and only if it is a procedure that is not generic and that has no formal parameters.

The Alsys DOS Ada Compiler imposes no additional ordering constraints on compilations beyond those required by the language.

Section 11

Limitations

11.1 Compiler Limitations

- The maximum identifier length is 255 characters.
- The maximum line length is 255 characters.
- The maximum number of unique identifiers per compilation unit is 2500.
- The maximum number of compilation units in a library is 1000.
- The maximum number of Ada libraries in a family is 15.

11.2 Hardware Related Limitations

- The maximum amount of data in the heap is limited only by available memory.
- If an unconstrained record type can exceed 8192 bytes, the type is not permitted (unless constrained) as the element type in the definition of an array or record type.
- A dynamic object bigger than 4096 bytes will be indirectly allocated. Refer to ALLOCATION parameter in the COMPILE command. (Section 4.2 of the *User's Guide.*)

INDEX

END